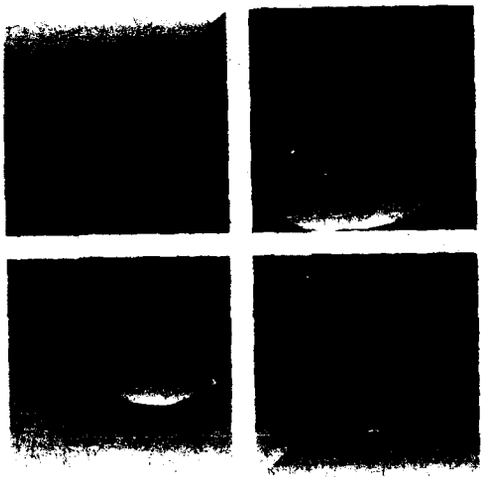
M91-35

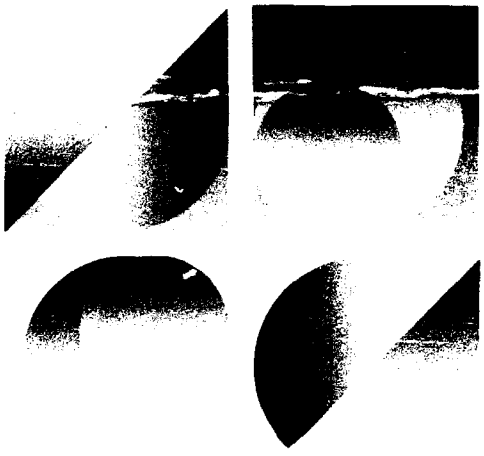
The Importance of Architecture in DOD Software



Dr. Barry M. Horowitz

M91-35

The Importance of Architecture in DOD Software



Dr. Barry M. Horowitz

July 1991

The MITRE Corporation
Burlington Road
Bedford, MA 01730

PREFACE

DOD's automated systems are likely to face more varied military threats than in the past that will require the ability to make system changes rapidly. In addition, defense budgets are likely to continue to decrease. It is important then, for both mission effectiveness and cost savings, that these systems be built with as much flexibility as possible to incorporate new capabilities and new technology. This paper proposes that an increased focus on digital system architecture can markedly improve system flexibility as well as yield significant cost savings.

The author, Dr. Barry M. Horowitz, is President and Chief Executive Officer of The MITRE Corporation.

ACKNOWLEDGMENTS

Many people at MITRE contributed data and ideas to this document. Special thanks are expressed to Judith A. Clapp, Dr. Richard J. Sylvester, and Gerard R. Lacroix.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION: A NEW DIRECTION FOR DOD SOFTWARE ACQUISITION

Our world is changing. The military threat to the United States posed by the Soviet Union for nearly fifty years is diminished, but there are new threats from rapidly evolving Third World countries that require rapid changes to military systems. Crises such as the recent events in the Persian Gulf highlight the need for flexible systems that can be changed quickly to meet the military's unanticipated challenges. In addition, the defense budget continues to be reduced — the government has less money to spend on systems.

The answer to this dual challenge — to make systems more flexible and to reduce the cost of defense systems — lies in the design of the digital system architecture, which includes the composition of hardware and software components, the structure that interconnects them, and the rules by which they interact. All too often, both government and industry focus too narrowly on achieving the initial requirements for systems and give little thought to being able to adjust to what the system may be required to do five or ten years later, or to what happens as hardware may no longer be supportable or advanced technology may become available for incorporation into the system.

Architecture design is the key to achieving the cost savings and operational flexibility inherent in digital systems. If the system is properly structured, then hardware components can be added or upgraded without expensive changes to the rest of the system. A good architecture allows a system designed to counter one threat to counter a different threat through localized modifications to the software that change the functional capability of the system or allow it to interoperate with other systems. What is more, under the right circumstances, these changes can be made very quickly.

DOD SOFTWARE: MORE IMPORTANT — AND MORE EXPENSIVE

The Value of Software

Software provides modern defense systems with a flexibility that cannot be achieved in any other reasonable way. Operation Desert Storm provides several excellent examples.

Patriot is an Army corps-level missile system primarily designed to counter aircraft. Given the inherent capability of the missile itself, the designers gave some thought to employing it to shoot down incoming enemy missiles. The Scud attacks during the war, however, focused everyone's attention on this threat with much more urgency.

Patriot's designers developed a new software package that increased the Patriot's effectiveness to counter the Scud threat. When radar tracks began to show that the Scuds were breaking up on re-entry, the designers further tuned the package to recognize and attack the Scud warhead, and not the debris that accompanied it.

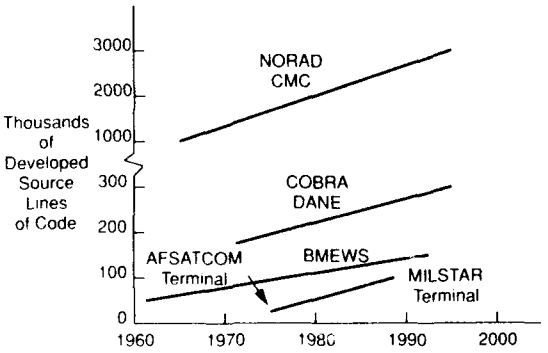
Without the modified software, Patriot would have been less effective. Yet the designers were able to implement this capability quickly and at a surprisingly low cost. No new missiles or radars were required. The software improvements could go to the war region in a briefcase.

Another example of this flexibility also involves Patriot, though at the much higher level of command, control, communications, and intelligence (C^3I). To improve Patriot's ability to react to the Scud attacks, which proceeded at much higher speeds than the targets normally confronting Patriot, U.S. space satellites were redirected to watch for Scud launches. When a launch was detected, the satellite relayed the targeting cues over a satellite link to the appropriate Patriot battery, leading to a successful interception. Minor software modifications permitted a network to be set up.

There are other, less dramatic examples of the value of software's inherent flexibility that came out of Desert Storm. Navy attack aircraft had been set up for years to attack Soviet targets, either at sea or on land. A cassette data tape provided the attack computers with the information they needed to launch their stunningly accurate attacks on targets that had only recently been identified.

Software was also the key to the effectiveness of Air Force jamming aircraft. Programmed for operations against Soviet-bloc radars, the jammers were faced with a mixture of Soviet, French, British, and Italian equipment. Software changes enabled the equipment to perform its task against this new threat far more quickly — and less expensively — than could have been done otherwise.

Precisely because of its flexibility, the DOD is buying more software in its systems and implementing functions in software that had previously been performed in hardware. Figure 1 shows this trend in a number of systems. For example, the latest version of the Cobra Dane radar system uses more software than did the original release, and the new Milstar terminal uses more software to perform more functions than did its predecessor, AFSATCOM. Desert Storm demonstrated that the flexibility software offers us is real and of great value to the military. It will become more so if we continue to have crisis scenarios that are a lot harder to predict and cause us to apply our systems in unplanned ways.



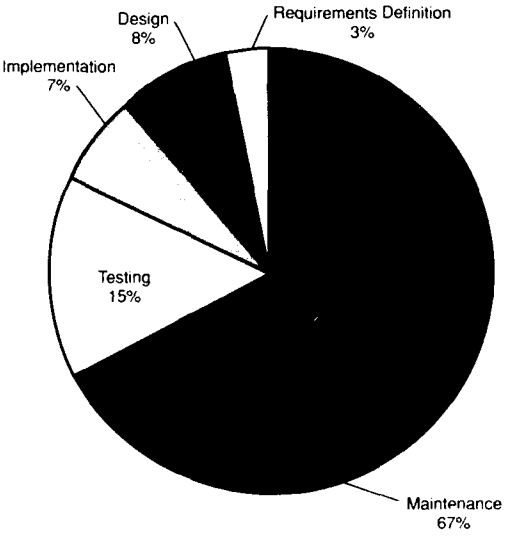
Source MITRE Analysis

Figure 1. Growth of C' Software Size

The Cost of Software

Since the DOD has been buying more and more software, its total expenditure on software has been increasing, and software is expensive. With shrinking military budgets, we have to find ways to use more software and yet reduce its cost.

Typically, two-thirds of what is spent on software is believed to be spent after the system becomes operational, during the maintenance phase, as illustrated in figure 2.

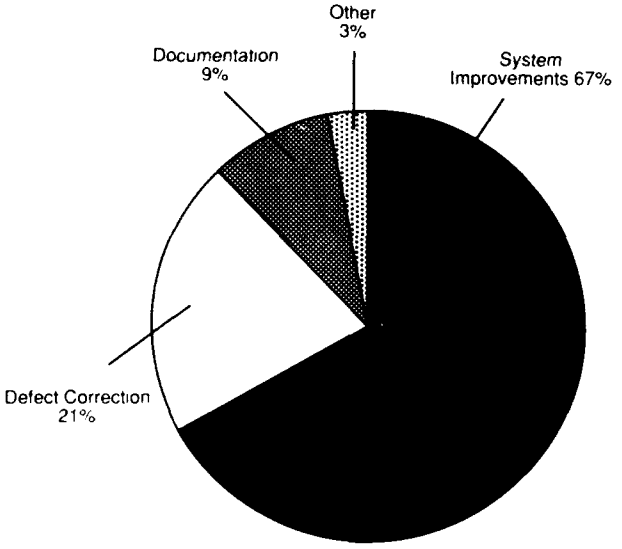


Source: Arthur, 1988

Figure 2. Software Life Cycle Cost Distribution

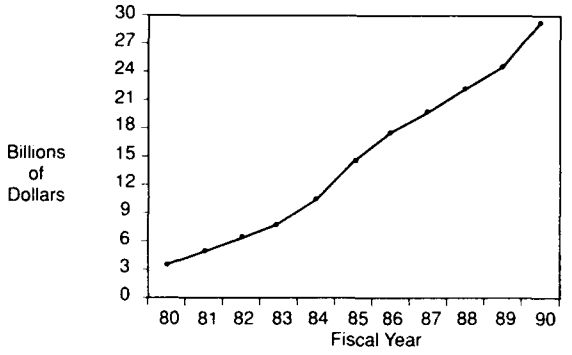
Looking at the distribution of software maintenance activities is itself illuminating. About two-thirds of the software maintenance effort for a system is typically spent on modifying the original system to provide new capabilities and to add new technology, at least twice the effort spent on making repairs. Figure 3 confirms this data for an Army command and control system.

Taking these two sets of data together suggests that about 45 percent of the effort spent on software is used to change the system after it has been deliv-



Source: Day

Figure 3. Software Maintenance Activities



MITRE estimate based on: Bureau of Economic Analysis Input-Output, GNP, and DOD expenditure data; 1987 Survey of Current Business, Department of Commerce; Census of Service Industries, Department of Commerce. Handbook of Labor Statistics, Department of Labor; and Census of Population, Occupation by Industry Matrix.

Figure 4. DOD Software Expenditures

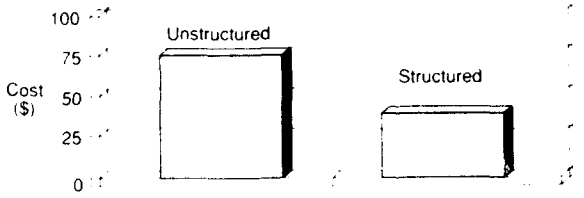
ered. Experience also shows that we often spend part of the system development effort making changes in response to changing or better understood requirements. We probably spend more than 50 percent of our software effort to change the capabilities of a system over its developmental and operational lifetime.

If we can design software systems to take only half as much effort to modify, we can reduce the life cycle cost of the entire software system by 25 percent. When applied to the total amount the DOD spends on software, this improvement can yield enormous cost savings.

While it is difficult to determine accurately how much the DOD spends on software, MITRE staff made a rough analysis that indicates the total amount to be approximately $30 billion per year (see figure 4). If we can in fact reduce the life cycle cost of software by 25 percent, the total savings will range between five and eight billion dollars every year.

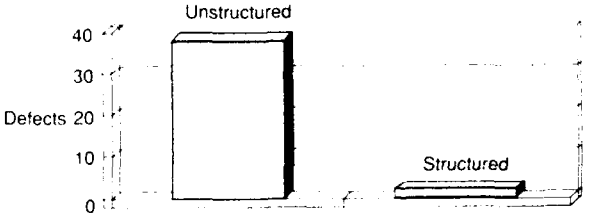
The example in figure 5 illustrates how these savings might be possible. Three thousand lines of new code were required to be added to a system of 50,000 lines. When the changes were made, the cost, time, and number of defects found in the delivered system were measured. Then, the structure of the software was improved, and the changes were again made. It cost only half as much to modify the structured software and it took less than half the time. As an added benefit, there were about one-eighth the number of errors in the structured software.

Another indication of increases in productivity that may accrue from well-structured software is shown in figure 6. The points on the graph represent software size and productivity for development of some systems programmed in Ada. One of those systems, the Command Center Processing and Display System Replacement (CCPDS-R), was developed with special attention to designing a system architecture and tools to facilitate its modification. The original system was then significantly modified to produce two new versions. Productivity data for the two modified versions of the system are shown in boxes in figure 6. The high overall productivity is due in part to the architecture that accommodated these changes and in part to tools that facilitated making changes. Further benefits were realized because the architecture made general system services more accessible and consequently the application modifications were smaller than they might otherwise have been. The productivity data were adjusted for the reused and tool-generated software.



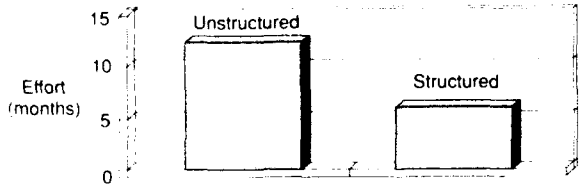
Source: Rock-Evans, Hales

Figure 5A. Structure Versus Cost to Change



Source: Rock-Evans, Hales

Figure 5B. Structure Versus Defects



Source: Rock-Evans, Hales

Figure 5C. Structure Versus Time to Change

This is even more impressive when the usual negative relationship between productivity and system size is taken into account.

While important, the dollar cost of making changes to the system is only one concern; time is another. Operation Desert Storm provided many examples of how well the flexibility of software served the allied cause, but there were also cases where we were not able to exploit software as we

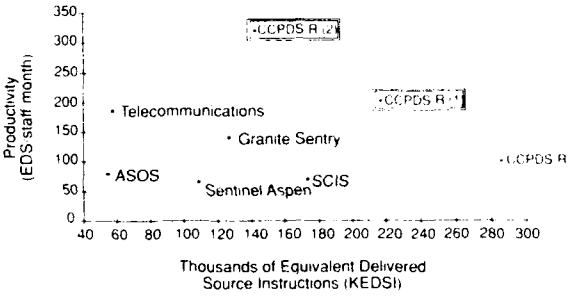
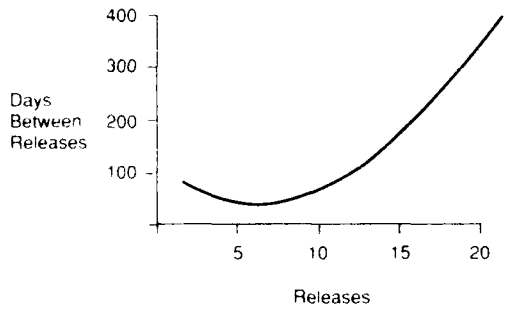


Figure 6. Ada Development Versus Modification

would have liked. Requests for changes to systems were made early in the campaign, and estimates were provided that said it would take 18 months to make the desired changes. This was obviously unacceptable, and the military found it hard to understand why it should take so long, given that software is supposed to offer great flexibility.

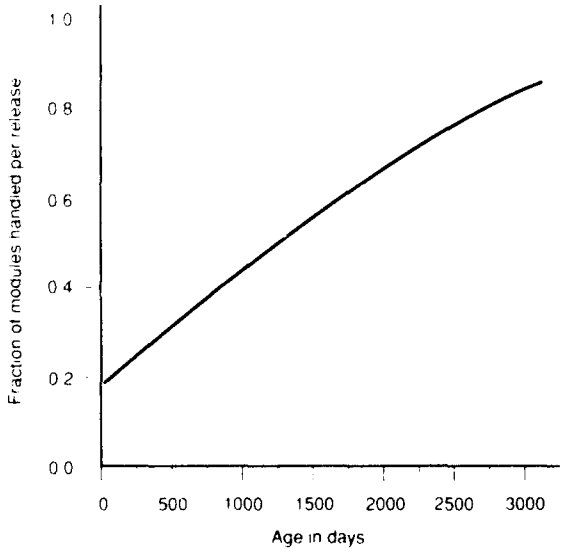
Software does provide flexibility, but it must be designed from the start with an architecture that allows it to do so. Furthermore, everyone concerned must preserve the integrity of the architecture; otherwise, flexibility can be lost through the process of change. As an example, figures 7A and 7B are plots of the time it took to create each release of an IBM operating system and the number of modules affected in each release. The graphs show a progression; that is, it took longer and longer to modify the system as the system grew older. This was due to the growing complexity of the system — more and more modules had to be changed for each new release. The software structure degenerated, which made it more difficult to determine which modules had to be repaired. In addition, the pattern of regression testing had to be more extensive since so many parts of the system had been affected by the modifications.

This complexity and uncertainty translates into more time and money, and this process begins a vicious circle — modifying the system makes the next modification even more difficult, time-consuming, and expensive.



Source: Lehman, Belady

Figure 7A. Release Interval Versus System Age



Source: Lehman, Belady

Figure 7B. Increasing Complexity with Age

ARCHITECTURE: THE INVISIBLE COMPONENT

The DOD does not usually buy architectures — it buys systems that meet explicit functional and performance requirements specified by the user or the acquisition agent. In most cases, the DOD does not ask for an architecture to be delivered; it should therefore come as no surprise that very few architectures are delivered.

This is not to say that the DOD does not receive a system architecture. Every system has some form of architecture, but the architecture the DOD receives may be quite convoluted and inflexible by the time the system moves from concept to fieldable implementation. The DOD does not specifically buy an architecture because there are no explicit specifications for its characteristics, no formal tests of its capabilities, and no formal control of its structure to prevent arbitrary changes once it has been defined. This is one reason why architecture is fundamentally invisible — operational users are not often aware that an architecture is even present if it does not directly affect the functional capabilities they are using.

Yet architecture is the main determinant of a system's characteristics. The efficiency of the system, and thus its performance, depend on how the architecture handles resource utilization; architecture determines how the system sustains operations when parts of the system fail. The architecture also determines how maintainable the system is; that is, 1) how much effort is required to find and fix errors; 2) how easy it is to add new capabilities through software; and 3) how much is required to move the software to different computer hardware. Although they may be invisible to the user, these characteristics, which are all determined by architecture, are very visible to developers and maintainers who must modify and add to the operational capabilities of the system.

If the DOD wants to buy architectures, it will first have to know how to ask for them, specify them, test them, demonstrate them, and prevent them from degenerating; in short it will have to perform all the operations that it performs now when buying other products.

In addition, DOD must perform a new task that is currently not part of its acquisition strategy — maintain explicit control of the architecture for the life of the system. One way of accomplishing this is to add architecture to the other aspects of the system that are controlled by the configuration

management system. Since the maintenance phase contains a large fraction of the system's software costs, the ultimate maintainer of the system — and thus, the government — must eventually assume control of the architecture. This will require a significant change in the way the government currently views architecture and its importance.

Architecture: A Definition

There is no single, commonly accepted definition of a digital system architecture. In the broadest sense, architecture is a system or style of building having certain characteristics of structure. When applied to digital computer systems, architecture includes the hardware and software components, their interfaces, and the execution concept that underlies system processing.

The simplest level of system architecture defines how the hardware and software that make up the system are partitioned into components, and how software components are assigned to hardware components. Figure 8 is an oversimplified example (only primary functions are shown) of a fighter aircraft's federated hardware and software structure, which consists of separate computers networked on a standard bus with individual software functions assigned to the individual computers. At this level, the defense industry generally does a fairly thorough job of understanding architecture, mainly because developers need to understand how much hardware of which types they need to buy.

Figure 9 is another view of the digital system architecture for the same aircraft, showing both the application software in the previous figure and the software that performs system-wide functions. The functions can be described as grouped into layers; in this view, software in any layer may utilize software only in its own layer or the layer below it. The computers in the lowest layer represent the segregation of hardware from software to increase their independence and to enhance software portability. This is an example of the first part of the definition of architecture — the arrangement of hardware and software components (namely, the structure).

The second element in the definition deals with the interfaces among key elements — for example, data communications according to a standard protocol (MIL-STD-1553). All computer-to-computer messages in the aircraft's avionics architecture must use this protocol; hence, adding new computers and new functions to the system is relatively simple (from a communications perspective) as long as the data bus has the needed capacity.

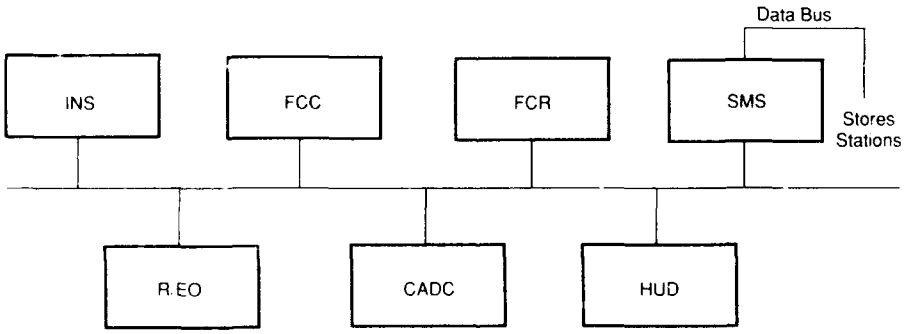


Figure 8. Hardware and Software Structure

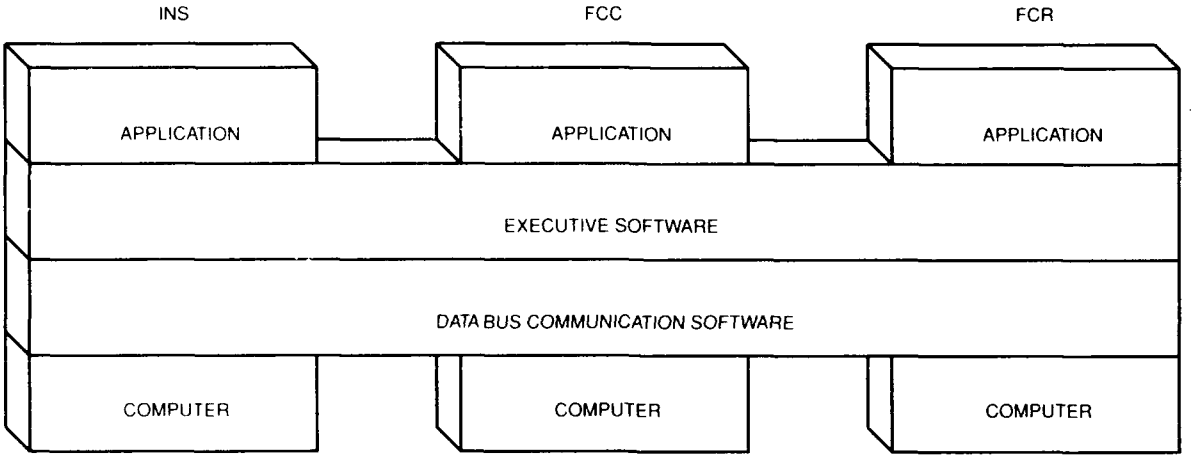


Figure 9. Digital System Architecture

The third element in the definition of architecture is the execution concept. In the sample avionics system previously shown, this concept is based on the cyclic execution of each function, precisely timed to repeat the computation on a planned schedule.

Taking structure, interfaces, and execution concept together produces one definition of architecture.

Of course, different vendors interpret the software part of the architecture in different ways. Figure 10 illustrates an IBM view of software architecture. In many cases, commercial companies can provide off-the-shelf components for the general system capabilities of DOD systems; in addition, groups of commercial hardware and software vendors are defining standard interfaces among layers and components. These open system architectures may provide the flexibility necessary to integrate, with a minimum of effort and system disruption, new hardware and software components with improved capability or maintainability. For example, the International Standards Organization (ISO) Open Systems Interconnection reference model defines the functions of each layer and the protocols for peer-level layers of a communications interface. Standards of this type permit the upgrading of elements of the system at particular layers without requiring the alteration of elements at other layers.

Figure 10 also illustrates the concept of service layers in the part of the architecture that is developed uniquely for one class of application (such as command centers or communications systems). These or other services must include error detection and recovery, interprocess communications, scheduling, and synchronization of processes. At this level of architecture, we must rely on the applications designers for standards within their design, as well as the quality control procedures to assure adherence to their standards.

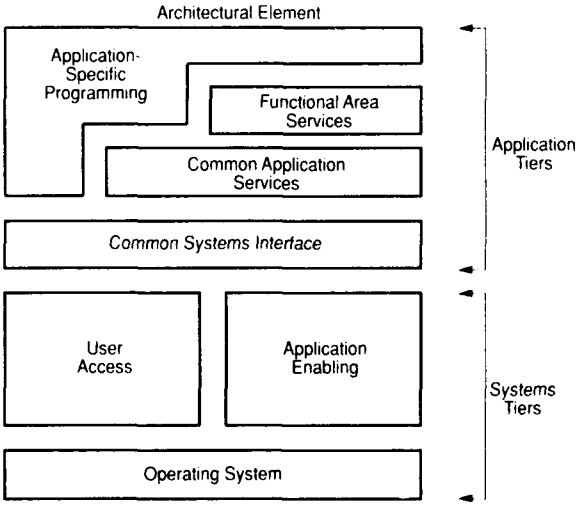


Figure 10. IBM View of Software Architecture

Architecture: Ramifications

The lack of a good architecture has a serious bearing on the cost, effectiveness, and availability of DOD systems. For many applications where high reliability and availability are necessary, the architectural concepts must incorporate failure management as well as other mission requirements. Trouble follows when this is not part of the initial architectural design.

Error handling is a critical component of any system, since errors will inevitably occur. Most systems have software to detect errors and to recover from an error when it is detected (for example, when a numerical value goes beyond expected bounds or when an operator pushes the wrong button). Given the critical nature of most DOD systems, it is crucial to keep the system in operation when errors occur. When we leave it to each programmer who has developed a part of a system to determine how to handle errors, the result is an unintegrated set of sometimes widely varying procedures that are often completely incompatible and even dangerous.

Recently, MITRE scanned the software for a large, safety-critical command and control system, and identified over 200 instances in the code that handled errors incorrectly. In many cases, the system detected the errors and then ignored them, or passed them to another part of the system that could not handle them. What was missing was a consistent, coherent, system-wide error-handling strategy, a critical attribute of architecture. Furthermore, there was no method of ensuring that individual programmers adhered to the failure management standards that should have been established with the architecture.

Data flow diagrams can show the execution concept of the architecture of a system (see figure 11). In this view, the sequence of processing, and which hardware and software components are involved as specific data moves through a system, are apparent. This end-to-end view of the system's treatment of an external input is called a string; in actuality, there are many levels of detail that can be represented by a hierarchy of data flows for the same string. A string is useful for assuring users that the system will perform the right functions on their data; it is also useful for estimating and controlling the time the system will take to respond to an input. What complicates the design of an architecture to meet response times is the large number of such strings that may be awaiting execution at the same time (as when many sensor reports are received or must be transmitted) and the contention over which string will use shared resources such as computers and communications lines.

To understand the timing aspects of a system, it is often necessary to develop a simulation that models the system architecture and the load on hardware and software components or to execute benchmark software on the actual hardware. The validity of the results depends on how accurately the load, the data flows, the hardware speed and capacity, and the timing of individual processes are represented in the model — even the most elaborate model yields useless results if the parameters are not accurate. The designer of the architecture must be given accurate information to design the architecture and to evaluate its performance; in other words, it is essential that there be good communications between modelers and architects or designers.

Since the demands on the hardware resources will change over time, the architecture must provide the flexibility necessary to upgrade hardware to faster or larger processors to accommodate requirements for increased processing loads or faster response time. Similar increases in bandwidth may be necessary in communications hardware to provide for increased loads. Models that correspond to an architecture can be useful in planning for and evaluating the effect of changes in the hardware configuration of a system architecture to meet new demands.

The Complexity of Architecture

Perhaps the main reason that we fail to address these different aspects of system architecture lies in the increasingly complex nature of the systems we build. Figure 12 illustrates the top level system architecture of the Joint Surveillance Target Attack Radar System, or Joint STARS. The actual architecture includes many more computers, many different data busses, and a large number of other components

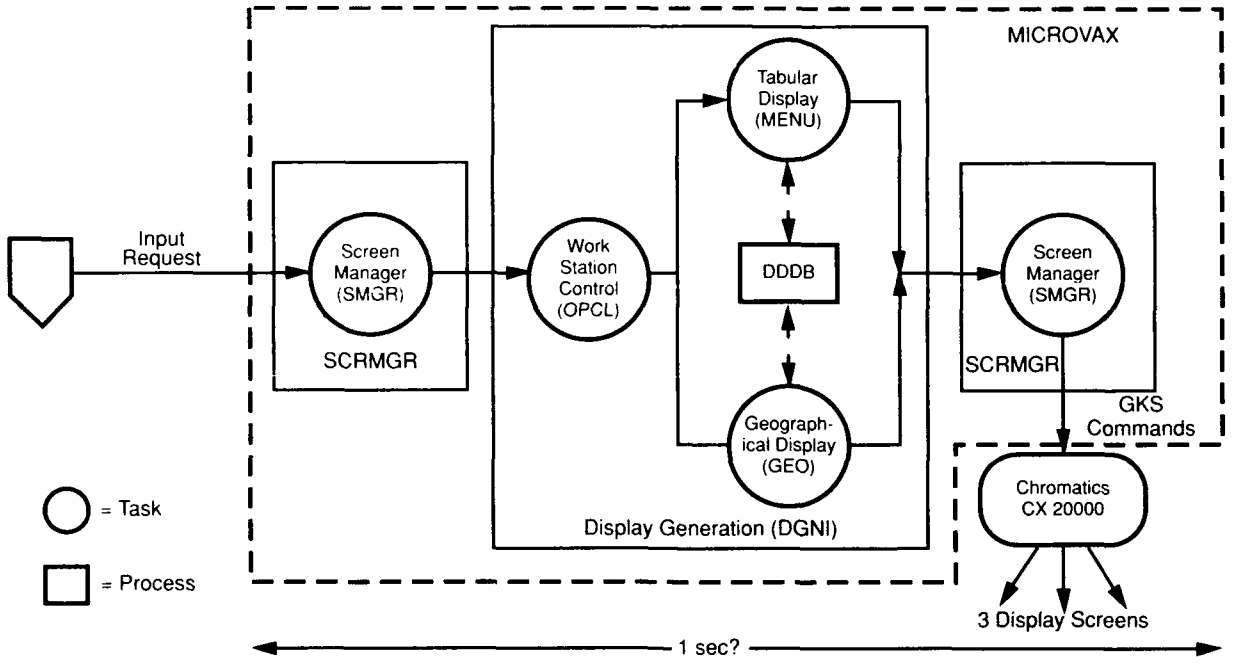


Figure 11. Data Flow Reference Model

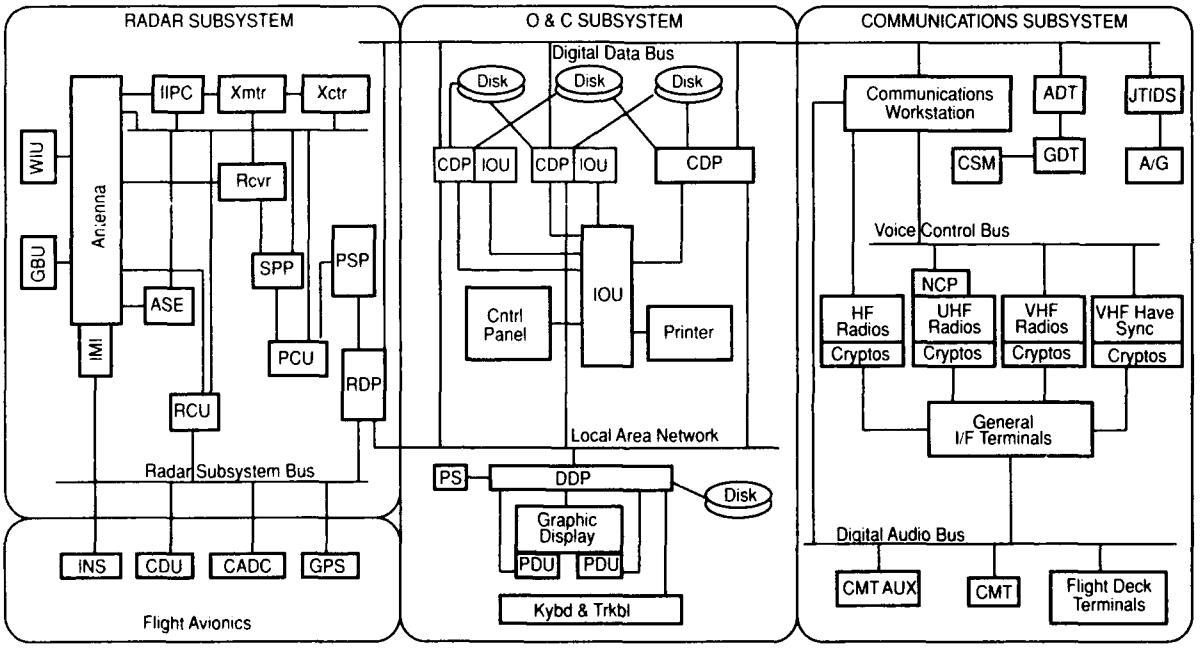


Figure 12. Joint STARS System Architecture

(not shown in the figure) to perform its demanding mission. The result is a large, complicated system that makes it difficult for developers to consider the many different aspects of architecture.

At the same time, the larger and more complicated the system, the more important good structure becomes. Developing and maintaining structure may be very difficult in a system of such complexity, but the rewards for doing so are even greater. These rewards include higher quality during the initial development, lower life cycle software costs, and the increased likelihood that the system will remain in operation far longer (due to its greater flexibility and ease of upgrading). Furthermore, the reuse of known and expandable architectures will reduce the amount of new software that has to be developed and increase the quality of the systems that use them.

ARCHITECTURE: THE WAITING SOLUTION

Technical Focus

At the start of a development program, when considering architecture pays the greatest dividends, the technical focus in the typical DOD program is often not on architecture. Rather, functional and performance requirements are generally focused on by both DOD and the contractor (refer to figure 13).

This lack of attention to architecture occurs because the government expresses its requirements in terms of specific, measurable system functions and performance requirements that matter to the immediate user, and not in terms of flexibility, which matters to the maintainer and next-generation user. Government standards, such as DOD-STD-2167A, require proof that a design satisfies all functional requirements, not that it is adaptable to change. Design documentation and reviews track individual system and software components, with less attention on the overall architecture until the components are integrated.

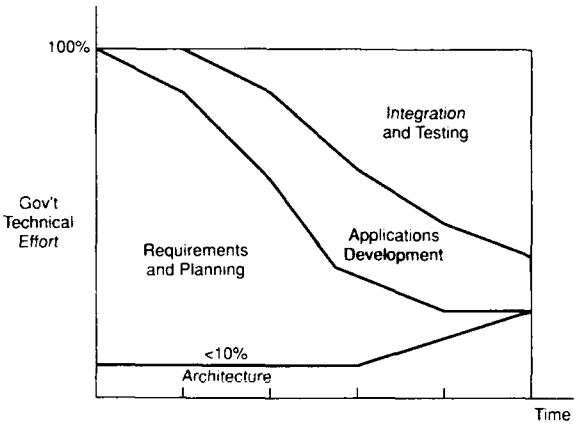


Figure 13. Technical Focus (Estimated)

As the figure shows, the failure to consider architecture throughout the program's development has serious ramifications as time goes on. The performance and control problems described earlier begin to mount, and the contractor is often forced to call on Red Teams and other desperate measures to modify the original architecture. Since it is done in haste and then only to allow the product to meet the specifications, this last-minute change in architecture does little to ensure the necessary efficiency and flexibility, and usually results in further degeneration of the basic design.

Faulty Emphasis

Both government and industry typically put almost all their efforts into the initial performance and functionality of a program in spite of the fact that these will change substantially over the life of the system. At the same time, there is a near-total lack of attention to an architectural baseline that would form a stable foundation for incorporating the system's changing requirements.

What we do ask for does not address the important architectural issues. For example, we state that the system shall be modular but don't state a good way to partition it into modules that will allow future expansion and change.

We also specify requirements for system growth in an ineffective way that does not relate to operational capabilities such as adding new message types or increasing message traffic. Timing and sizing margins — for example, half the time and twice the memory — cannot ensure that the resources provided are allocated in such a way that they can be used to meet future requirements.

With the advent of distributed systems, timing and communications bandwidth margins become important in providing for future growth. The government needs to assure that growth is expressed in operational terms, and not just in physical terms.

Because of the government emphasis on meeting immediate requirements within schedule and cost, even industry perceives that the government is not seriously interested in controlling maintenance costs. In a 1990 Air Force Scientific Advisory Board study of software maintenance, 123 businesses were asked what the government thinks is important when awarding software contracts:

Overall project cost	6.2 out of 7
Proposed product performance	5.5
Contractor experience in area	5.5
Timeliness	5.3
Last contract an advantage	4.8
Project software development cost	4.6
Contractor software capability	4.4
Ease of software maintenance	**3.4**
Software maintenance cost	**3.3**
Software portability	2.9

Their view of the government's stress on cost and system performance, rather than long term maintenance, is readily apparent.

Commercial Architecture

It has recently become evident that commercial software users have become more concerned with architecture. As users become familiar with vendors' capabilities, their expectations increase. In turn, many software vendors are now providing software interface standards that enable interoperability across different computer hardware families and allow users to pick and choose among competing software vendors.

These commercial architecture trends can do nothing but help DOD software efforts, because DOD is a large buyer of software and hardware that support these interoperability standards. Even embedded, special-purpose militarized systems rely heavily on commercial systems to assist in software support. The DOD cannot try to take the lead because these standards are driven by the commercial marketplace; however, the DOD can use to advantage the opportunities in the commercial market for open architecture standards. Unfortunately, these commercial standards cannot include the service standards that are heavily application-dependent; these must be left to the application designer to establish and implement.

Availability of Tools

The commercial market is also in the lead in providing tools that support the designer in generating and documenting architectures. There are tools to enable developers to analyze the linkage between different software modules, the control flow, the flow of data, and the timing of the various operations, and to assess and improve architectures.

Many tools can only perform their analyses after the software is already written. These tools can still be used to understand what has been developed and to evaluate how easily it can be modified, before it is fielded or later. The investment may be small, and the potential payoff large. The following table lists some representative examples of available tools:

Name	Vendor	Analyzes
Logiscope	Verilog	Module structure, path coverage
ACT/BAT	McCabe	Flow graphs, structure graphs
ADAS	CADRE	Dynamic behavior, timing
STATEMATE	i-Logix	Structure, dynamic behavior
CPN	Meta	Dynamic behavior, simulation
Adagen	MarkV	Ada static structure, dynamic behavior
CARDtools	Ready	Timing threads
TAGS	Teledyne, Brown	Dynamic behavior, simulation

The government must acquire these tools and use them if it is going to buy architectures and understand them.

In addition to commercially available tools, project-specific tools can improve the productivity of software development for a specific architectural design. Referring to the CCPDS-R program again, the contractor developed a tool to automatically generate the communications software that linked applications. The applications programmer needed only to list the elements of data that were required from each application and were necessary to each application. The tool used this information to generate efficient and correct communications following a standard pattern.

RECOMMENDATIONS: FINDING AND APPLYING ARCHITECTURE

Good architecture potentially can provide significant cost savings as well as greatly increased system flexibility. To obtain these benefits, we must put architectural requirements in system specifications, emphasize the early satisfaction of these architectural requirements, give contractors incentives to use proven architectural concepts, and control the architectural configuration over the life cycle of the system. We believe this can be done.

System Specification

Since contract requirements drive the entire development of a system, the surest way to ensure adherence to a sound architecture is to put architectural requirements in the system specification. This does not mean that the specification will define the exact architecture to be used, but rather that it will specify what the architecture is to do. In cases when the application domain is well understood and a sound architecture is already available, the government may find it in its best interest to be more restrictive than in other situations lacking such a clear precedent.

To specify accurately the criteria architectures must meet, we must also determine how to qualify them. There are few measures of system designs that accurately predict flexibility and expandability. We will have to depend on a combination of techniques, including demonstrations that the system can be modified as well as analyses of features of the architecture. We are beginning to establish relationships between measurable features and rate of errors as well as ease of change. As figure 14 shows, the more calls a module makes on other modules, the more errors occur.

Early Satisfaction of Architectural Requirements

To reap the maximum possible benefit from architectural requirements, we should specify that contractors cannot write large amounts of applications software until they have developed an architecture that the DOD has evaluated and approved. The only applications software that would be written before this point would be that necessary to help evaluate the architecture and reduce other serious risks, not to perform the actual task at hand. We can no longer afford the risk of developing architecture and applications concurrently; on the other hand, if contractors have successful architectures and control procedures that they have used before, they can use them again. In fact, the quality and effectiveness of a

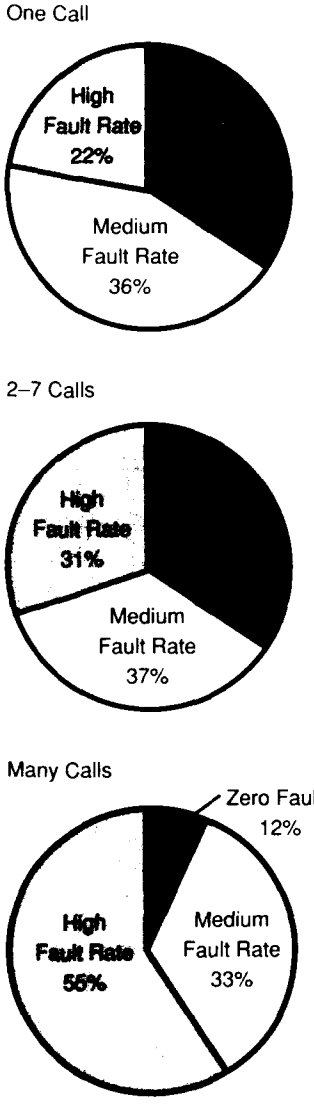


Figure 14. Effect of Control Structure on Errors

previous architecture as well as the tools available to support development of applications within the architecture should be an important factor in the selection of contractors on a program.

We should also control these architectures after we evaluate and approve them. Changes would be weighed against the need for future flexibility throughout the life of the program.

Contractor Incentives

Contractors will have to be given incentives to change from their current emphasis on meeting immediate requirements to a longer term view. They will have to set up their own controls to keep applications software writers from corrupting the architecture; in other words, during development, contractors will have the architecture under configuration control. Rules and standards have to be defined as part of the architecture. Tools should facilitate the integration and modification of components within the architecture so we know that the standards of the architecture are observed.

Contractors who have good architectural awareness should be treated better than those who do not. The development community needs to start working on architecture with the software maintainers to ensure that we deliver to them whatever is necessary for them to sustain and use the architecture.

Getting Started

It is recommended that the DOD put together a government and industry team to develop specification and contractual language for buying architectures. Approaches for evaluating and testing architecture need to be agreed upon as well. We are confident that this can be done and we have begun to develop an example. This team should also see that we use the experience that we have acquired on programs to determine what the contractors and the government have done to create good architectures, and to define the criteria for evaluating architectures.

We also need to consider buying single architectures for closely similar clusters of systems to reduce the cost of buying and maintaining unique architectures for each. For existing systems, we must work to introduce architectural improvements without disrupting operational use of the systems.

It is crucial that industry participate as part of the team that would create the specification language and evaluation criteria. The insight of a joint government-industry working group on architecture will be of considerable benefit to the DOD during this time of changing missions and increased need for flexible systems.

REFERENCES

Arthur, L. J. *Software Evolution, the Software Maintenance Challenge*. New York: John Wiley & Sons, 1988.

Card, D. N. *Measuring Software Design Quality*. Englewood Cliffs, N.J.: Prentice-Hall, 1990.

Day, R. *A History of Software Maintenance for a Complex U.S. Army Battlefield Automated System, Proceedings of the Conference on Software Maintenance*. New Jersey: IEEE, 1985.

DOD-STD-2167A, Defense System Software Development, 1988.

Lehman, M. M., and Belady, L. A. *Program Evolution — Processes of Software Change*. New York: Academic Press, 1985.

Rock-Evans, R., and Hales, K. *Reverse Engineering: Markets, Methods, and Tools*. England: Ovum, Ltd., 1990.

United States Air Force Scientific Advisory Board. *Report of the Ad Hoc Committee on Post-Deployment Software Support*. U. S. Government Printing Office, 1990.